SOLUTION BRIEF

ENDOR LABS

Open Source Security For Python And AI Apps

Gain visibility into your hidden Python dependencies and prioritize reachable, exploitable risks.

When Log4j happened, every organization scrambled to figure out if it was in their code. Finding vulnerable Python dependencies is substantially harder. Are you ready?

DID YOU KNOW?

Python is the language of choice for creating AI models, but because of this popularity, it's likely to be targeted by malicious parties. Unfortunately, detecting vulnerable dependencies is difficult because many Python dependencies aren't in the manifest so they aren't caught by traditional SCA tools.

Python applications can use dependencies not declared in manifest files. What happens when those "phantom dependencies" are vulnerable and you can't see them?

Traditional SCA Tools - Like Snyk - Can't Find All Python Dependencies

Setting up an AI/ML tech stack with Python requires manually setting up dependencies such as TensorFlow, Torch, PyTorch and sklearn in the environment (think compatibility with the OS and CUDA drivers) where model training is going to take place. This can lead to these dependencies being provided that aren't declared in a manifest. Discrepancies between what the project uses and what is declared in the manifest can happen when you:

- Add a dependency to the virtual environment
- Allow the platform to provide packages and their versions
- Remove packages that are no longer used
- Allow direct usage of transitive dependencies

Problem #1
Higher Risk of Breaches

Cause: False Negatives
The SCA tool fails to identify artifacts that contain vulnerable code because it can't find dependencies that aren't in the manifest.

This limits the utility of the SCA tool while simultaneously providing your organization with a false sense of security.

Problem #2
Wasted Engineering Cycles

Cause: Noise
The SCA tool wrongly thinks an artifact contains vulnerable code because it surfaces unused dependencies based on the manifest.

The engineering team is stuck with tracking down whether a package is used and justifying the findings.

Endor Labs scans Python source code to find hidden dependencies and uses reachability analysis to prioritize just the risks that can actually impact your application.

1. Source Code as a Ground Truth

As the primary "source of truth", Endor Labs uses the source code offers the clearest insight into which dependencies are called upon and used.

2. Correlate with Package Manager Data

After establishing dependencies, the data is cross-referenced with package manager information, identifying phantom and unused dependencies.

3. Correlate with the File System

By comparing dependencies declared by package management manifests with those used in the code and those available in the file system, you get a complete picture of the dependencies used.

4. Highlight Discrepancies

Any variations between the actual code and package manager definitions are clearly marked, alerting developers to potential issues like missed vulnerabilities or unnecessary packages.

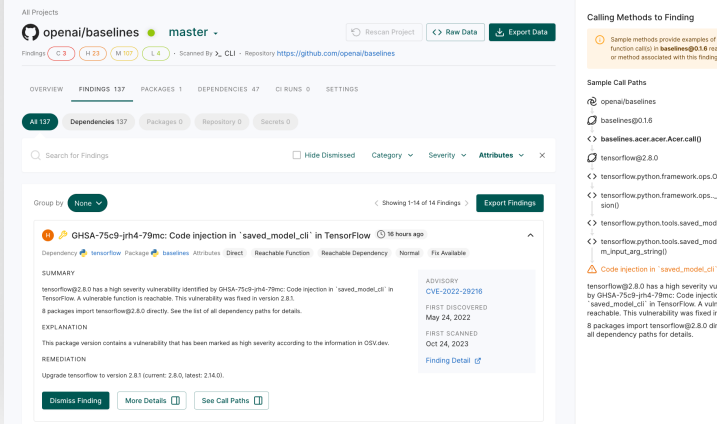
OpenAI Baselines Case Study

To demonstrate the difference between traditional SCA tools and Endor Labs, we'll look at OpenAI Baselines (a set of implementations of reinforcement learning algorithms).

Endor Labs' proprietary dependency resolution finds 47 dependencies for this package, while traditional tools find just the 11 that are defined in the package's manifest file. Note that these tools do not discover TensorFlow in the manifest as it is a provided dependency.

Endor Labs discovered 129 vulnerabilities, as opposed to the 1 that manifest-based scanners tools find. 128 of these affect TensorFlow (the package no one else can find). When reachability analysis is applied, that list is reduced by almost 90%, down to 14 reachable dependencies with all critical vulnerabilities marked as unreachable. In this example, we show how we find all relevant vulnerabilities (limit false negatives) that impact a project and then help prioritize the findings to cut the noise to a manageable number of actionable findings.

Traditional SCA tools (like Snyk) miss this risk because the vulnerable dependency is not declared on the manifest!



Developers quickly understand and remediate this Python dependency risk.

- **What is the vulnerability?** A code injection in `saved_model_cli` in TensorFlow
- **How was the vulnerability introduced?** It was pulled in by TensorFlow 2.8.0
- **Is the vulnerability exposed to users?** Yes, there is a call path through tensorflow@2.8.0
- **Is there a fix?** The vulnerability was fixed in TensorFlow 2.8.1
- **How can I remediate?** Upgrade to TensorFlow 2.8.1 or newer